

CRC

Basic idea: the data is a binary polynomial. Divide this polynomial by another, the remainder is the CRC checksum.

Ex: $101110 \rightarrow x^{10} + x^8 + x^4 + x^2$

Mathematically, we want to construct a value C based on M which is evenly divisible by a special polynomial G .
 In Exercise 25 it's done using binary arithmetic with no carry.

For instance suppose $\frac{M}{G} = Q \text{ rem } R$, which means that

M divided by G gives a quotient Q + a remainder R

equivalently: $\frac{M}{G} = Q + \frac{R}{G}$ or $M = GQ + R$

From this, we can see that $M-R$ will be evenly divisible by G :

$$M-R = GQ + R - R$$

$$M-R = GQ$$

$$\frac{M-R}{G} = Q \text{ rem } 0$$

If we transmit $M-R$, the receiver can divide by G : if it divides evenly, we assume there was no error. If there is a non-zero remainder, there must have been an error.

However, M is not recoverable from $M-R$ unless we know R , + we don't know R unless we know M . Therefore we can detect (some) transmission errors, but we haven't actually transmitted the message.

What we would like to do is send both M and R , the receiver can do the subtraction themselves & see if $M-R$ is evenly divisible by G .

We can do this by concatenating M & R together. Mathematically, this means shifting M up by however many bits are in R & then adding R to the result. But once again, if the receiver doesn't know R , he doesn't know how many bits to shift, & therefore cannot reconstruct M .

However, we do know that R is strictly less than G , so we can agree that we will always treat R as having the same number of bits as G , just zero filling it as needed.

In summary:

- Assume G has W bits
 - Multiply M by 2^W , which is the same as appending W bits w/ value 0 to the end of M . Call this augmented message E ($E = M \cdot 2^W$)
 - Divide E by G using binary arithmetic w/o carries.
 - The Quotient Q is ignored.
 - Subtract the remainder R from E to get C .
- $$C = E - R = (M \cdot 2^W) - R$$
- Send C

As we already saw, C will necessarily be divisible by G . Notice that on this arithmetic, addition & subtraction are the same, so we can equivalently say $C = E + R$. Finally, note that since R will not have more than W significant bits, & since $E = M \cdot 2^W$ has all W low bits all set to 0, C is equivalent to M w/ R (padded to W bits) appended to its end.

The receiver therefore checks for errors by seeing if the received version of C , \tilde{C} , is only divisible by G . If it is, they assume no errors ($\tilde{C} = C$), & can recover M by subtracting R & dividing by 2^w .

$$M = \frac{(\tilde{C} - R)}{2^w}$$

Or equivalently, they just cut off the last w bits.

Note that R is not the remainder of $\frac{C}{G}$, it is the remainder of $\frac{M \cdot 2^w}{G}$, which is not necessarily the same.

Division

In this arithmetic, two numbers w/ the same number of significant bits cannot be compared for size, because addition and subtraction are the same. For instance:

$$1011 + 101 = 1110$$

$$1011 - 101 = 1110$$

More to the point: $1001 - 110 = 1111$, so 1001 cannot be less than 1111 . On the other hand, $1111 - 110 = 1001$, so 1111 cannot be less than 1001 .

Therefore, two numbers w/ the same number of significant bits can always be divided one into the other w/ a non-zero quotient.

$$\begin{array}{r} 1011 \overline{) 11110} \\ \underline{- 1011} \\ 0101 \end{array} \rightarrow \frac{1110}{1011} = 1 \text{ rem } 101$$

$$\begin{array}{r} 1110 \overline{) 1011} \\ \underline{- 1110} \\ 0101 \end{array} \rightarrow \frac{1011}{1110} = 1 \text{ rem } 101$$

However, a number w/ fewer significant bits is always less than one w/ more sig bits:

$$1011 > 101$$

Therefore, a number w/ more sig bits divides a number w/ fewer sig bits 0 times (with a remainder).

So we do division just like long division, but the subtraction is binary everywhere:

$$\begin{array}{r}
 000011000010 \\
 10011 \overline{) 1101011011000} \\
 \underline{-10011} \\
 010011 \\
 \underline{-10011} \\
 000001 \\
 \underline{-000001} \\
 000010 \\
 \underline{-000000} \\
 000101 \\
 \underline{-000000} \\
 001010 \\
 \underline{-000000} \\
 010100 \\
 \underline{-100111} \\
 001110 \\
 \underline{-000000} \\
 \boxed{01110} \text{ remainder}
 \end{array}$$

Recall that we don't care about the quotient, only the remainder. Take a look at the steps we followed during the division, specifically w.r. to the remainder.

At each step, if the remainder's highest order bit is in the same spot of the divisor (they have the same number of significant bits), then the divisor will go in once, which means we are going to end up subtracting the divisor from the current remainder.

On the other hand, if the remainder's highest bit is lower than the divisor's, it goes in 0 times, & the remainder is left unchanged.

Then, either way, we shift the next bit from the dividend into the remainder.

Now, if the divisor is G & has w significant bits, the dividend is an array of bits M , & the remainder is R , we have:

```

R = 0;
for (bit in M) {
    if (R >> (w-1)) { // check for bit w set on R
        R = R ^ G;
    }
    R = (R << 1) | bit;
}

```

That's the basic straightforward division algorithm. Besides for CRC we need to shift M up by w bits before doing this division.

Table based implementation

Consider what happens as we process 2 bits of the message of our remainder register currently looks like:

$$\boxed{a_3 \mid a_2 \mid a_1 \mid a_0}$$

(using a 5 bit poly $g(x)$)

When we're ready for the next message bit, m_i , a_4 will determine whether or not we XOR $g(x)$ into the register:

$$+ (a_4 * \boxed{a_3 \mid a_2 \mid a_1 \mid a_0 \mid m_i} \quad \boxed{g_4 \mid g_3 \mid g_2 \mid g_1 \mid g_0})$$

$$\boxed{a_3 + (a_4 g_4) \mid a_2 + (a_4 g_3) \mid a_1 + (a_4 g_2) \mid a_0 + (a_4 g_1) \mid m_i + (a_4 g_0)}$$

We'll call these bit values: $\boxed{b_4 \mid b_3 \mid b_2 \mid b_1 \mid b_0}$

Now the next bit, it will be $b_4 = a_3 + (a_4 g_4)$ which determines whether or not we XOR:

$$+ (b_4 * \boxed{b_3 \mid b_2 \mid b_1 \mid b_0 \mid m_{i+1}} \quad \boxed{g_4 \mid g_3 \mid g_2 \mid g_1 \mid g_0})$$

$$\boxed{c_4 \mid c_3 \mid c_2 \mid c_1 \mid c_0}$$

$$\text{Notice that } c_0 = m_{i+1} + (b_4 g_0) \\ = m_{i+1} + ((a_3 + (a_4 g_4)) g_0)$$

$$\text{while: } c_1 = b_0 + (b_4 g_1) \\ = m_i + (a_4 g_0) + ((a_3 + (a_4 g_4)) g_1)$$

$$\text{Similarly: } c_2 = b_1 + (b_4 g_2) \\ = a_0 + (a_4 g_1) + ((a_3 + (a_4 g_4)) g_2)$$

$$C_3 = b_2 + (a_4 g_2) \\ = a_1 + (a_4 g_2) + ((a_3 + (a_4 g_1)) g_2)$$

and

$$C_4 = b_3 + (a_5 g_3) \\ = a_2 + (a_5 g_3) + ((a_3 + (a_4 g_1)) g_3)$$

So you can see now that after shifting in 2 message bits, each of our four bits values is of the form:

$$C_j = x_j + (a_4 g_{j-1}) + ((a_3 + (a_4 g_1)) g_j)$$

where x_0 is m_{in} , x_1 is m_0 , & $x_j = C_{j-2}$ for $2 \leq j \leq 4$
 & note that g_{-1} (used for C_0) is simply 0.

This can be further "simplified" as

$$C_j = x_j + T_j$$

where T_j is only dependent on g and on the top two bits of a ($a_1 + a_2$).

So, in the end, we have:

$$+ \begin{array}{|c|c|c|c|c|} \hline a_2 & a_1 & a_0 & m_0 & m_{in} \\ \hline \end{array} \\ + \begin{array}{|c|} \hline T \\ \hline \end{array} \\ \begin{array}{|c|c|c|c|c|} \hline C_4 & C_3 & C_2 & C_1 & C_0 \\ \hline \end{array}$$

Recall that T is a function of a_1, a_2 & g (all of g).
 This means we can precompute T for all possible values of $a_1 + a_2$, for a given g . Once we have that, all we need to do is shift over two bits of message into R from the right, & XOR it with $T(a_1, a_2, g)$.

A more easy way to write this is $T_g(a_1, a_2)$ which makes T_g a four element table of values that get XOR'd into the register, and then values are selected by the pair of bits $a_1 + a_2$.

The reason T_g is dependent on a_1 & a_2 is because we looked at shifting 2 message bits in before XOR'ing a value from T_g . If we had shifted in a third bit (message) before doing the XOR, then a_2 would have shifted off the register and affected the XOR value as well. In that case, we would have had:

$$\begin{array}{c}
 \begin{array}{|c|c|c|c|c|c|} \hline a_1 & a_2 & a_3 & \text{message} & a_{n-2} & a_{n-1} \\ \hline \end{array} \\
 + \quad \begin{array}{|c|c|c|c|} \hline T_g(a_1, a_2, a_3) \\ \hline \end{array} \\
 \hline
 \begin{array}{|c|c|c|c|c|c|} \hline a_n & a_3 & a_2 & a_1 & a_0 & \\ \hline \end{array}
 \end{array}$$

T_g can be made for various # of bits. T_g for S bits will of course have 2^S entries. Each entry is the same size as the polynomial G_p . The bigger S , the faster you can process data (S bits at a time), but the bigger the table is. That means you need more memory, & to make larger to simulate the table.

The limit for S is that S cannot exceed k , the size of G_p . If you try to shift more than k bits at one time in registers, then the leading message bits will get shifted off the way out & end up affecting the XOR value. Technically, you could do this, but you would need to construct the address from both the registers & the appropriate bits of the message.

Computing the table

You can see by reviewing the previous diagrams that the table entry $T_g(x)$ is what you get if you load the top 5 bits of the remainder register with x , and fill the rest of the register with 0's, then perform the usual algorithm 3 times. The value in the register is $T_g(x)$.

Eg, if $S=3$ & $x = a_2 2^2 + a_1 2 + a_0$:

Start with

a_2	a_1	a_0	0	0
-------	-------	-------	---	---

Then do 3 iterations:

$$+ (a_2 * \begin{array}{|c|c|c|c|c|} \hline a_1 & a_0 & 0 & 0 & 0 \\ \hline \end{array}) \quad ①$$

$$\begin{array}{|c|c|c|c|c|} \hline b_2 & b_1 & b_0 & g_1 & g_0 \\ \hline \end{array}$$

$$\downarrow$$

$$\begin{array}{|c|c|c|c|c|} \hline b_1 & b_0 & g_1 & g_0 & 0 \\ \hline \end{array}$$

$$+ (b_2 * \begin{array}{|c|c|c|c|c|} \hline b_1 & b_0 & g_1 & g_0 & 0 \\ \hline \end{array}) \quad ②$$

$$\begin{array}{|c|c|c|c|c|c|} \hline c_2 & c_1 & c_0 & g_2 & g_1 & g_0 \\ \hline \end{array}$$

$$\downarrow$$

$$\begin{array}{|c|c|c|c|c|c|} \hline c_1 & c_0 & g_2 & g_1 & g_0 & 0 \\ \hline \end{array}$$

$$+ (c_2 * \begin{array}{|c|c|c|c|c|c|} \hline c_1 & c_0 & g_2 & g_1 & g_0 & 0 \\ \hline \end{array}) \quad ③$$

$$T_g(a_2 2^2 + a_1 2 + a_0) \rightarrow \begin{array}{|c|c|c|c|c|c|} \hline d_2 & d_1 & d_0 & g_3 & g_2 & g_1 \\ \hline \end{array}$$

Table Modification

Recall that before computing the checksum, we need to append 0 bits to the message. This is equivalent to shifting an even bit bits that are all 0 at the end of the calculation.

However, just as we saw in deriving the table based algorithm, shifting in 0 bits has the effect that the top K bits already in the register will produce a "Carry" value that depends only on those K bits in the generator polynomial G. This value is XOR'd into the register.

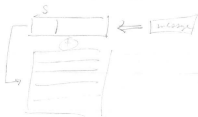
In the case (the first 0's), the bit, so if all of the bits in the register that are shifted out and produce the value that is XOR'd. That means that when we do the XOR, the entire register is 0's and so the XOR has the effect of putting the computed value in the register.

In other words, the purpose of shifting in the final 0's is to force the last bit message bit all the way through the register, so they go all the way through the algorithm.

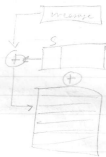
What this means is that message bits never actually need to be in the register; their only purpose is to control how the generator poly. G gets shifted + XOR'd onto itself. This is because at the end, this same value gets XOR'd into the reg., but the register has been shifted to be all 0's.

So we can modify the algorithm, instead of shifting data into the register, only to have it XOR'd with some value, then shifted out to select a value from T_0 , we can just XOR the next S bits of the message with the top S bits from the register. (as they are shifted out), & use that to select from T_0 .

So it used to look like this:



Now it looks like this:



And in this way, message data is processed by the algorithm immediately, instead of needing to be shifted through the register, so we can skip the final $W-1$ bits. This is the only point of this change, is to skip the last $W-1$ bit-rows.

Initial Values

We saw in the last section that as message bits shift out of the top of the register (in the straight-forward algorithm), they construct the final value by determining whether or not the poly is XOR'd into the register.

But we didn't discuss what is supposed to happen before data bits enter the top of the register. This is the first "block" of the algorithm.

The answer is that it depends on the initial value of the register. In the pure polynomial division interpretation of CRC, the register is initially 0, which means the first w bits of message data substituted any w 2's are simply shifted out. Since they are 0, we don't XOR the poly into the register, so these first w rounds don't do anything except to shift the message data in, which we've already gotten used to by XOR'ing the data up to the top of the register.

However, in real world applications, the register is often preloaded w/ some non-zero value, that uses the first w bits that are shifted out as the straight-forward algorithm are not all zero & will cause 0, so we XOR'd in. However, as we've seen before we can determine the result of processing all of these w bits in one go, & XOR everything else into it. Since "everything" is getting XOR'd into the register, we can simply initialize the register to the other value instead of the "original" initial value.

The initial value used in the straight-forward algorithm is called the "indirect" initial value. The modified value that we can use w/ our modified fast-forward algorithm is called the "direct" initial value.

Converting indirect to direct is simply a matter of feeding the indirect initial value through the straight-forward algorithm. I.e., load the registers w/ the indirect values, shift out 1 bit at a time, & XOR G_1 into the current register value if the shifted bit is 1.

E.g. initial value is 1011 (indirect).

G_1 is 1001

$$\begin{array}{r}
 \begin{array}{r}
 10100 \\
 1001 \overline{) 10110000} \\
 \underline{100111} \\
 1000 \\
 \underline{1001} \\
 000100 \\
 \uparrow \\
 \text{(direct)}
 \end{array}
 &
 \begin{array}{r}
 10100 \\
 \times 1001 \\
 \hline
 10100 \\
 00000 \\
 00000 \\
 + 10100 \\
 \hline
 10110100 \\
 + 0100 \\
 \hline
 10110000 \checkmark
 \end{array}
 \end{array}$$

This of course is just calculating the CRC of the indirect value.